Enhancing the Performance of the Hungarian Algorithm with GNN Warm-Starts

Jad Agbaria Muhammed Mhamed
Supervisor: Ilay Yavlovich

November 2025

Abstract

The Linear Assignment Problem (LAP) is a foundational combinatorial optimization task with applications across vision, logistics, scheduling, and tracking. Classical solvers such as the Hungarian (a.k.a. Munkres) algorithm guarantee optimality with $O(n^3)$ worst-case complexity. However, for large problem sizes ($n \geq 10^3$) and challenging cost distributions, latency becomes a bottleneck. We present a **learning-augmented** approach that preserves optimality while delivering practical speedups: a lightweight neural model (**OneGNN**) predicts **row dual variables** u, and the corresponding column duals v are recovered by a **dual-feasibility "min-trick"** $v_j = \min_i(C_{ij} - u_i)$. These duals seed a modified LAPJV/Hungarian solver (**seeded LAP**), reducing augmentations and iterations. We evaluate on synthetic families (Uniform, Sparse, Block/Metric) and sizes 512–16384 with 70/15/15 splits. **Our method achieves** $\sim 1.6\times$ **end-to-end speedups** over a standard cold-start Hungarian baseline while maintaining **100% optimality** of the final matching (guaranteed by feasibility and fallback). We detail the theory, system design, and experimental methodology to reproduce and extend these results.

1 Introduction & Motivation

The Linear Assignment Problem (LAP) involves finding a permutation π for a square cost matrix $C \in \mathbb{R}^{n \times n}$ that minimizes the total cost $\sum_i C_{i,\pi(i)}$. This problem is central to numerous applications, including task assignment in operations research, frame-to-frame data association in object tracking, resource scheduling in logistics, and bipartite matching in machine learning pipelines such as recommendation systems or graph alignment.

Classically, the Hungarian (Munkres) algorithm provides an exact solution with $O(n^3)$ worst-case time complexity, and efficient implementations like those in SciPy or the LAPJV variant are readily available. However, for large n—particularly beyond a thousand—and for cost matrices with adverse structures (e.g., ambiguous or near-uniform costs that create many near-ties), the practical runtime can become prohibitive, limiting use in real-time or high-throughput settings. This tension is well documented in applied LAP workloads, where runtimes are highly sensitive to problem size and solutions are required under tight latency budgets [5].

These observations motivate the question of whether machine learning can reduce the computational work required by the solver without compromising optimality. We propose a warm-start strategy that predicts dual variables in a way that ensures they are dual-feasible, making them safe to initialize an exact solver. Although the asymptotic complexity remains $O(n^3)$, warm starts can significantly reduce constant factors by providing an initial set of dual variables that render many reduced costs zero from the outset. The resulting density in the equality graph allows the solver to match a large portion of rows greedily, reducing the number and length of augmenting paths. In practice, this translates to fewer iterations and less computational work, especially in instances where cold starts would require extensive dual adjustments.

Furthermore, by focusing on predicting only the row duals and recovering column duals via the min-trick, we ensure that the warm start is not only efficient to compute but also inherently feasible, preventing any violation that could lead the solver astray. This approach leverages the structure of the primal–dual formulation to inject learning where it most effectively reduces effort, bridging classical optimization with modern neural methods.

2 The Linear Assignment Problem (LAP)

Problem statement. Given a square cost matrix $C \in \mathbb{R}^{n\times n}$, the LAP asks for a one-to-one assignment (a permutation) π that minimizes the total cost

$$\min_{\pi\in S_n} \sum_{i=1}^{n} C_{i,\pi(i)}. \tag{1}$$

Equivalently, let $X \in \{0,1\}^{n\times n}$ be a *permutation matrix* with $X\mathbf{1} = \mathbf{1}$ and $X^\top\mathbf{1} = \mathbf{1}$ (every row/column has a single 1). Then

$$\min_X \langle C, X\rangle \quad \text{s.t.} \quad X\mathbf{1} = \mathbf{1},\ X^\top\mathbf{1} = \mathbf{1},\ X \geq 0,\ X \in \{0,1\}^{n\times n}. \tag{2}$$

LP form (for duality). We will use the linear-programming relaxation only to introduce dual potentials, no integrality facts are needed in what follows. Relaxing $X \in \{0,1\}^{n\times n}$ to $X \in [0,1]^{n\times n}$ yields

$$\min_X \langle C, X\rangle \quad \text{s.t.} \quad X\mathbf{1} = \mathbf{1},\ X^\top\mathbf{1} = \mathbf{1},\ X \geq 0. \tag{3}$$

Graph view. LAP is the minimum-weight perfect matching on the complete bipartite graph $K_{n,n}$: left nodes $U = \{1,\dots,n\}$, right nodes $V = \{1,\dots,n\}$, and edge (i,j) has weight C_{ij}. The assignment corresponds to selecting n vertex-disjoint edges with minimum total weight.

2.1 Duality, reduced costs, and optimality certificates

The linear program (LP) (3) has the dual

$$\max_{u,v} \sum_{i=1}^{n} u_i + \sum_{j=1}^{n} v_j \quad \text{s.t.} \quad u_i + v_j \leq C_{ij} \quad \forall\, i,j, \tag{4}$$

where $u \in \mathbb{R}^n$ and $v \in \mathbb{R}^n$ are the row and column potentials. For any dual-feasible (u,v), define the reduced cost of edge (i,j) by $r_{ij} := C_{ij} - u_i - v_j \geq 0$, the set of equality (tight) edges is

$$E_0 = \{(i,j) : r_{ij} = 0\},$$

and the corresponding equality graph is $G_0 = ([n],[n],E_0)$.

Two standard facts:

- **Optimality certificate (strong duality).** If X is feasible for (3) and (u,v) is feasible for (4) with $\langle C, X\rangle = \sum_i u_i + \sum_j v_j$, then both X and (u,v) are optimal.

- **Complementary slackness.** In any optimal solution, $X_{ij} > 0$ implies $(i, j) \in E_0$, in particular, all matched edges lie in E_0.

Hence, it suffices to search for a perfect matching inside G_0, once found, it is optimal for the original LAP.

2.2 Classical algorithms and complexity

State-of-the-art exact algorithms (Hungarian/Munkres, Jonker Volgenant/LAPJV) operate on the reduced-cost landscape:

1. Maintain dual potentials (u, v) so that $r_{ij} \geq 0$ (dual feasibility).
2. Greedily build/repair a matching using only equality edges E_0.
3. When stuck, *decrease* certain reduced costs by adjusting (u, v) along an alternating tree (Dijkstra-like label updates) until new equality edges appear, then continue augmenting.

For dense costs, the practical (and worst-case) complexity is $O(n^3)$ time and $O(n^2)$ memory. For sparse graphs with $m \ll n^2$ edges, shortest-augmenting-path implementations exploit sparsity, with typical bounds expressed in n and m (e.g., $O(nm + n^2 \log n)$ in many variants). In all cases, the number of *augmentations* and the work per augmentation are driven by how many equality edges already exist.

2.3 Why matrix structure matters

Although the worst-case is $O(n^3)$, real instances vary widely:

- **Ambiguous/Uniform costs** produce few initial equality edges and many ties $\Rightarrow$ more dual adjustments and augmentations.
- **Structured (Metric/Block/Banded)** instances often start near a good dual, inducing dense E_0 near the true permutation $\Rightarrow$ fewer/shorter augmentations.
- **Sparse** costs (few viable edges) reduce search space but can be tricky if equality edges disconnect, performance depends on how quickly equality edges connect unmatched rows/cols.

This explains why *warm-starts* (good initial duals) can dramatically reduce runtime even though the asymptotic worst-case remains $O(n^3)$.

2.4 Numerical aspects and common variants

Building on the reduced-cost view above, practical normalizations and problem variants directly affect the initial density and numerical stability of the equality graph G_0 and thus the amount of work the Hungarian/JV solver performs. We list the conventions we use and their implications.

Preprocessing and scaling. Row/column shifting (subtracting per-row/column minima) preserves the optimal assignment and, up to roundoff, leaves E_0 unchanged, it improves numerical stability. Standard Hungarian/JV implementations effectively perform these reductions implicitly.

Rectangular and partial assignment. Rectangular $n \times m$ $(n \neq m)$ instances are reduced to square by padding with dummy rows/columns and appropriate costs (e.g., zeros to allow optional matches, or a large penalty M to forbid them). LAP also admits *partial* assignment where only $k < n$ pairs are selected.

Min vs. max, costs vs. profits. Maximization is handled by converting to costs via $C' = -C$ (or by subtracting from a large constant). All theory and algorithms carry over unchanged.

2.5 Takeaways for our method

The dual variables (u, v) and the equality graph E_0 govern the effort an exact solver expends. If we can predict u close to optimal and recover v by

$$v_j = \min_i \big(C_{ij} - u_i\big),$$

then (u, v) is dual-feasible, many edges have $r_{ij} = 0$, and a large fraction of rows can be matched immediately on E_0. The solver spends much less time creating equality edges and searching long augmenting paths. This is precisely the lever our learning-augmented warm-start exploits in later sections.

3 Classical LAP Solving Algorithms

This section introduces two classical exact algorithms for the Linear Assignment Problem (LAP): the Hungarian (also known as Kuhn–Munkres) algorithm and the Jonker–Volgenant algorithm (LAPJV). Both maintain dual feasibility and grow a matching on edges of zero reduced cost (the equality subgraph), adjusting potentials to create new tight edges. We present the Hungarian method through the classical zero-covering view and LAPJV through the modern successive shortest-augmenting-path perspective. These highlight how initial dual variables influence the number of relabel/augmentation steps and the overall computational effort.

3.1 Classical Kuhn–Munkres - The Hungarian Algorithm

This presentation works directly on the cost matrix and uses two kinds of marks on zeros: *starred* (chosen for the current partial matching) and *primed* (temporary candidates).

Notation. Let A denote the current working matrix (initialized as C). A *covered* row/column is temporarily ignored, an *uncovered* position is active.

1. **Row reduction.** For each row i, subtract its minimum:

   $$A_{ij} \leftarrow A_{ij} - \min_j A_{ij}.$$

   Every row now has at least one zero.

2. **Column reduction.** For each column j, subtract its minimum:

   $$A_{ij} \leftarrow A_{ij} - \min_i A_{ij}.$$

   Every column now has at least one zero.

3. **Initial starring and column cover.** Scan A, whenever you find a zero in a row and column that contain no starred zero yet, *star* it. After this pass, *cover* every column that contains a starred zero. If the number of covered columns is n, the starred zeros form a perfect matching. **stop**.

4. **Prime uncovered zeros until an augmenting path appears.** Repeat:

4.1. Find an *uncovered* zero, *prime* it.
4.2. If there is no starred zero in its row, go to Step 5, an augmenting path has been found.
4.3. Otherwise, cover this row and *uncover* the column containing the starred zero, continue searching for an uncovered zero.
4.4. If no uncovered zero exists, go to Step 6.

5. **(Augment) Build alternating path and flip stars/primes.** Starting from the newly primed zero, build an alternating sequence of *primed* and *starred* zeros by zig-zagging between rows/-columns (each primed zero's column contains a starred zero unless it is the start). *Unstar* every starred zero on this path and *star* every primed zero on the path. Then:
   - Erase all primes.
   - Uncover all rows and columns.

   Return to the column-cover test in Step 3. If n columns are covered, **stop**, otherwise continue from Step 4.
6. **(Shift) Create new zeros by shifting the uncovered minimum.** Let δ be the minimum value among all *uncovered* entries of A. Modify A:

$$A_{ij} \leftarrow \begin{cases} A_{ij} - \delta, & \text{if } (i,j) \text{ is uncovered,} \\ A_{ij} + \delta, & \text{if } (i,j) \text{ is covered twice (both row and column covered),} \\ A_{ij}, & \text{otherwise.} \end{cases}$$

   This preserves feasibility and produces at least one new uncovered zero. Return to Step 4.

Correctness intuition. Row/column reductions and the δ-shift emulate legal updates of dual potentials and preserve the nonnegativity of reduced costs. By Kőnig's theorem for bipartite graphs—stating that the size of a maximum matching equals the size of a minimum vertex cover—the minimum number of row/column covering lines that hit all zeros (a vertex cover in the zero graph) equals the maximum number of pairwise nonconflicting zeros (a matching) [10] Consequently, when the cover size reaches n, a perfect zero-matching exists. The star/prime machinery then reveals an augmenting path in the zero graph, increases the matching by one, and the process repeats until a perfect matching is obtained.

Complexity. With efficient data structures, this dense-matrix routine runs in $O(n^3)$ time and $O(n^2)$ memory.

3.2 Shortest-augmenting-path view (LAPJV / Jonker–Volgenant)

High-performance implementations maintain explicit *dual potentials* (u, v) and a partial matching M, and repeatedly find a cheapest augmenting path in the *reduced-cost* graph.

Reduced costs and equality edges. Define $r_{ij} = C_{ij} - u_i - v_j \geq 0$. Equality edges are $E_0 = \{(i,j) : r_{ij} = 0\}$. The algorithm maintains dual feasibility ($r_{ij} \geq 0$) at all times.

One augmentation (for a free row i).

1. Initialize Dijkstra-like labels for columns: `used`[j]$= 0$, `slack`[j]$= \infty$, and predecessor pointers.
2. Repeatedly:

   2.a. Among unused columns, pick j with minimum `slack`[j], let δ be that value.

2.b. **Update potentials:** for all visited rows/columns,

$$u_{i'} \leftarrow u_{i'} + \delta, \qquad v_{j'} \leftarrow v_{j'} - \delta, \qquad \texttt{slack}[j''] \leftarrow \texttt{slack}[j''] - \delta.$$

This decreases some reduced costs and creates new equality edges.

2.c. If j is *free* (unmatched), we have reached the end of an augmenting path, we have reached the end of an augmenting path, go to (c). Otherwise, follow its matched row and continue relaxing columns via reduced costs.

2.d. **Augment:** trace predecessors back to i and flip matched/unmatched edges along the alternating path, increasing $|M|$ by one.

Why it is the same algorithm. The potential updates above correspond exactly to the δ-shift in the classical procedure, the Dijkstra labels (`slack`) are the current shortest distances in the reduced-cost metric. Each augmentation increases the matching size by one until it becomes perfect. The overall complexity for dense costs is $O(n^3)$, but with a smaller constant and excellent practical performance.

3.3 Dual interpretation and link to our warm-start

Both variants can be seen as adjusting (u, v) to maintain $r_{ij} \geq 0$ and to *create* equality edges that enable augmentations. In our method, we *predict* a good u and recover v by

$$v_j = \min_i (C_{ij} - u_i),$$

which is dual-feasible by construction. Consequently, the equality graph E_0 is already dense around the optimal permutation, so the Hungarian/LAPJV solver needs *fewer and shorter* augmentations, yielding the observed $\sim 2\times$ end-to-end speedups while preserving exact optimality.

4 Graph Neural Networks for Warm-Starting LAP

4.1 Background: message passing on graphs

A Graph Neural Network (GNN) learns representations by repeatedly exchanging information over a graph. For a graph $G = (\mathcal{V}, \mathcal{E})$ with node features $h_i^{(0)}$, a generic T-layer message-passing GNN updates node i via

$$h_i^{(t+1)} \;=\; \phi^{(t)}\Big(h_i^{(t)},\; \underset{j \in \mathcal{N}(i)}{\mathrm{AGG}}\, \psi^{(t)}\big(h_i^{(t)}, h_j^{(t)}, e_{ij}\big)\Big), \tag{5}$$

where ψ produces messages along edges (optionally using edge features e_{ij}), and AGG denotes a permutation-invariant aggregator (e.g., sum/mean/max). Because the aggregator is invariant to neighbor ordering, GNNs respect graph symmetries (node permutations), which is essential for combinatorial problems.

4.2 Modeling LAP as a bipartite graph

The $n \times n$ cost matrix C induces a complete bipartite graph $K_{n,n}$ with left nodes $U = \{1, \ldots, n\}$ (rows) and right nodes $V = \{1, \ldots, n\}$ (columns). Each edge (i, j) carries the scalar feature $e_{ij} = C_{ij}$ (optionally normalized). A natural GNN formulation would run message passing across all n^2 edges to produce node embeddings for U and V, then decode assignment or dual variables. However, dense message passing costs $O(n^2)$ *per layer* in time and memory.

Our objective. We do not need a full assignment purely from the GNN. It suffices to produce *good dual hints* so that an exact solver finishes quickly. In particular, we aim to predict only the *row duals* $u \in \mathbb{R}^n$, then recover v via the dual-feasible *min-trick* (Section 7.2):

$$v_j \;=\; \min_i \big(C_{ij} - u_i\big),$$

which ensures dual feasibility ($u_i + v_j \le C_{ij}$) by construction.

Row-local features suffice because the dual u_i primarily depends on the cost distribution in row i relative to others, and statistics like min and std capture the "competitiveness" that determines how much "slack" u_i needs to accommodate the best assignments. Predicting only u biases the model towards row-wise reasoning, which aligns with the asymmetric nature of the min-trick and keeps the model lightweight, avoiding the overhead of jointly modeling u and v couplings.

5 Objectives & Contributions

Our goal is to reduce the wall-clock time of exact LAP solvers at scales where latency is a bottleneck, while preserving the optimal matching produced by classical methods. We approach this by predicting only the row dual variables u with a lightweight OneGNN, recovering the column duals v via the dual-feasible min-trick, and then handing these potentials to an exact seeded LAPJV/Hungarian solver. The central objective is therefore twofold: first, to maintain correctness by ensuring that (u, v) is always dual-feasible (and falling back to a cold solver if needed), second, to consistently reduce solve time across families and sizes by presenting the solver with a dense equality graph and short augmentations.

Our contributions are threefold. Conceptually, we frame learning not as a replacement for combinatorial search but as a principled warm start grounded in LP duality, which guarantees safety. Methodologically, we design a compact, row-local model with a single-pass feature extractor that keeps inference overhead negligible, and we integrate it into a seeded LAPJV that accepts (u, v) directly. Empirically, we establish approximately $2\times$ end-to-end speedups over a cold Hungarian baseline on synthetic families (Uniform, Sparse, Block/Metric) for $n = 512$–16,384, with 100% optimality of the final matching.

6 Related Work

Classical exact methods for LAP trace back to Kuhn [1] and Munkres [2], with high-performance shortest-augmenting-path implementations such as Jonker–Volgenant (LAPJV) [3]. . These methods are exact with $O(n^3)$ worst-case complexity, in practice their runtime depends on the reduced-cost landscape (density and connectivity of equality edges).

Learning-based solvers (approximate/differentiable). A line of work replaces the combinatorial search with a learned assignment module. GLAN converts the cost matrix into a bipartite graph and trains a deep graph network that labels edges, yielding a differentiable LAP "layer" used inside vision pipelines (e.g., MOT). These models optimize assignment accuracy and end-to-end learning rather than certificates of optimality, prior differentiable layers (Sinkhorn-like, CNN/RNN-based) are known to degrade as problem size grows, and GLAN reports improved robustness in that setting [5].

Learning-augmented warm starts (exact solver in the loop). Instead of replacing the solver, "learned duals" predict dual variables and then *project* them to feasibility in linear time before warm-starting Hungarian. Theory shows the running time depends on the ℓ_1-distance between the feasible seed and the optimal duals, and experiments demonstrate speedups on matching benchmarks [7]. Our approach follows this paradigm but makes a pragmatic simplification: we learn only the row duals u and recover v by the min-trick, removing the feasibility projection step.

Related matching problems. For Maximum-Weight Matching, recent evidence suggests that pure GNN heuristics may underperform simple greedy baselines on broad graph families, highlighting the value of exact solvers or provably safe warm starts when guarantees matter [9].

Our approach in context. Our contribution sits at this intersection: a *one-sided dual-prediction* warm start (learn u, compute v deterministically) that always yields a dual-feasible seed for LAPJV/Hungarian. On synthetic families (Uniform, Sparse, Block/Metric) and sizes $n = 512$–16,384, this design delivers *approximately* $\sim 1.6\times$ *end-to-end speedups* over a cold Hungarian baseline while maintaining *100% optimality* of the final matching. The learned component remains lightweight (sub-10% of total time for $n \geq 2$k), and the approach generalizes cleanly across cost families and problem sizes.

7 Technical Foundations

7.1 Primal–Dual Formulation

As introduced in Section 2, we adopt the LP relaxation (3) and its dual (4). We henceforth use the reduced costs $r_{ij} = C_{ij} - u_i - v_j \geq 0$, the equality set $E_0 = \{(i,j) : r_{ij} = 0\}$, and the equality graph $G_0 = ([n],[n],E_0)$ defined there, these will be our basic objects for warm-starting and analysis. **Dual feasibility** requires $u_i + v_j \leq C_{ij}$. **Complementary slackness** ensures that for matched pairs $(i, \pi(i))$, equality holds.

7.2 The "Min-Trick" for v

Given any candidate u, define

$$v_j := \min_i \big(C_{ij} - u_i\big).$$

Then for all i, j, $u_i + v_j \leq u_i + (C_{ij} - u_i) = C_{ij}$, so (u, v) is **dual-feasible** by construction. This property makes the warm-start **safe**: the exact solver can legally start from this dual and refine as needed.

Feasibility and safety. For any predicted u, define $v_j = \min_i(C_{ij} - u_i)$. Then $u_i + v_j \leq C_{ij}$ for all (i, j), so the pair (u, v) is dual-feasible. The seeded solver therefore starts from a valid dual certificate and can only improve (or confirm) the dual objective, if numerical screening deems the warm start unhelpful, we fall back to a cold solver, which guarantees that the final assignment equals the exact optimum. This argument explains why the method never sacrifices correctness.

Simplified example. Consider

$$C = \begin{pmatrix} 4 & 1 & 3 \\ 2 & 0 & 5 \\ 3 & 2 & 2 \end{pmatrix}, \quad u = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}.$$

Project $v_j = \min_i(C_{ij} - u_i) = (1, 0, 1)$. The reduced costs are

$$r_{ij} = C_{ij} - u_i - v_j = \begin{pmatrix} 2 & 0 & 1 \\ 1 & 0 & 4 \\ 1 & 1 & 0 \end{pmatrix} \geq 0.$$

Equality edges $(1, 2), (2, 2), (3, 3)$ already reveal a perfect matching supported on zeros. Starting from this dual, the Hungarian/LAPJV solver performs no dual decrease and only a short augmentation to obtain the optimum, illustrating how dense equality sets reduce work.

7.3 Why Warm-Starts Help

Hungarian/LAPJV iteratively seeks tight edges and augments along alternating paths. If initial (u, v) are close to optimal, many edges are already tight and fewer adjustments are needed, reducing the number/length of augmentations and total iterations.

8 Architecture & Design

8.1 Pipeline Overview

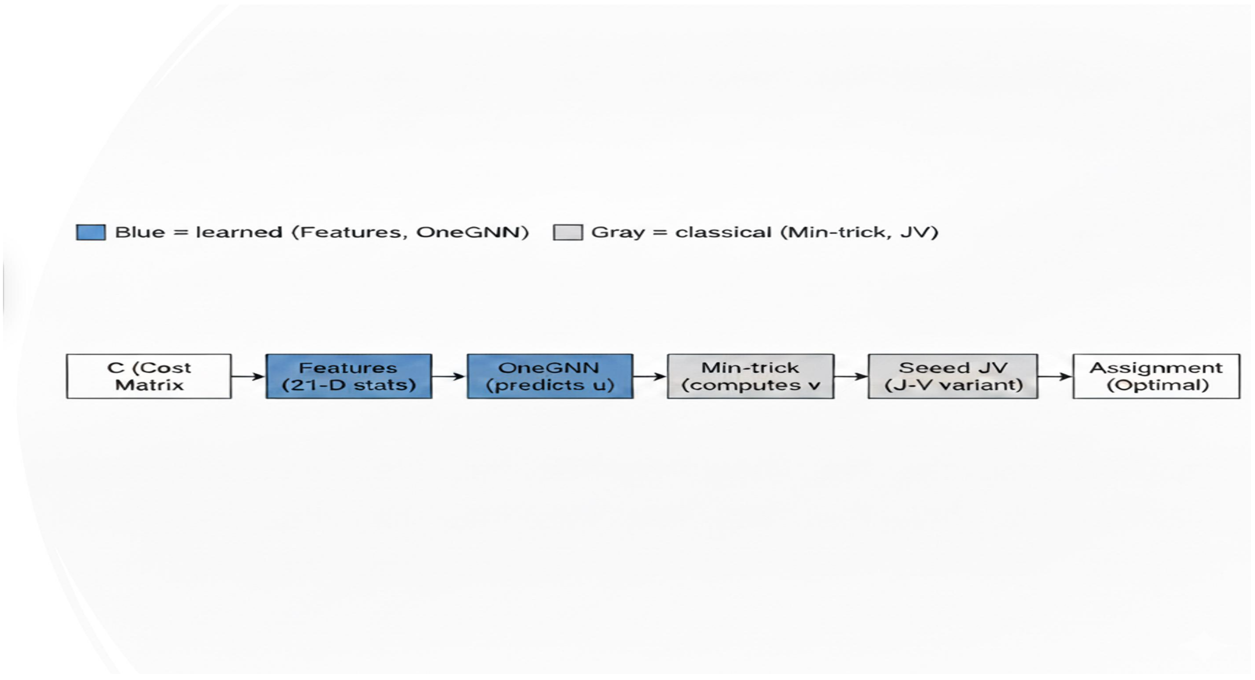


Figure 1: Warm-start pipeline. Blue: learned, gray: classical.

8.2 Row Feature Extractor (21-D per Row)

To enable efficient row-local processing in the *OneGNN* architecture while avoiding the $O(n^2)$ computational overhead of dense message passing across edges, we compress each row i of the cost matrix $C \in \mathbb{R}^{n\times n}$ into a fixed-size, informative feature vector $f_i \in \mathbb{R}^{21}$. This extraction is performed with a single pass over the row, ensuring $O(n)$ work per row (and $O(n^2)$ overall, matching the cost of reading C once). The design goal is to achieve high **information density** per row capturing local statistics, global competitiveness cues, and positional context, while remaining lightweight for scalability to large n (e.g., up to $n = 10^4$ in our experiments).

The 21-dimensional feature vector consists of 13 statistical and competitiveness features derived from the row's cost distribution, plus 8 positional encodings for the row index. These features are computed as follows (let C_i denote row i, with $m = n$ columns for square matrices, we use stabilized numerics with $\epsilon = 10^{-9}$ to avoid division-by-zero or overflow):

- **Location and spread metrics (6 features):** These summarize the central tendency, variability, and shape of the cost distribution in row C_i, providing a robust overview of the row's "difficulty" or cost landscape.
  - Minimum cost: $\min(C_i)$, the lowest cost in the row (indicates the best potential match).
  - Maximum cost: $\max(C_i)$, the highest cost (captures the row's cost range).
  - Mean cost: $\frac{1}{m}\sum_{j=1}^{m} C_{i,j}$ (average cost level).
  - Standard deviation: $\sqrt{\frac{1}{m}\sum_{j=1}^{m}(C_{i,j} - \text{mean}(C_i))^2}$ (measures spread around the mean).
  - Median absolute deviation (MAD): Median of $|C_{i,j} - \text{median}(C_i)|$ over j, clamped to $\geq \epsilon$ (robust outlier-resistant spread measure).
  - Softmax entropy: $-\sum_{j=1}^{m} p_j \log(p_j + \epsilon)$, where $p_j = \frac{\exp(-C_{i,j})}{\sum_k \exp(-C_{i,k}) + \epsilon}$ (stabilized with row-min shift for numerical stability, quantifies uncertainty or "flatness" in the cost distribution).
- **Competitiveness indicators (5 features):** These capture how "contested" or differentiated the row's best options are, helping the model infer assignment challenges without full graph access.
  - Best-vs-second gap (if $m \geq 2$): Sorted costs $S = \text{sort}(C_i)$, then $S_1 - S_0$ (absolute difference between the two lowest costs, larger gaps suggest easier decisions).
  - Normalized competition (if $m \geq 2$): $(S_1 - S_0)/(\max(C_i) - \min(C_i) + \epsilon)$ (relative gap, normalized by row span, values near 0 indicate high competition).
  - Top-k mean (with $k = \min(10, m)$): Mean of the k smallest costs in C_i (summarizes quality of top candidates).
  - Top-k std: Standard deviation of the k smallest costs (variability among top options).
  - Assignment difficulty (if $m \geq 2$): $\dfrac{1}{\left(\frac{1}{m-1}\sum_{j=1}^{m-1}(s_{j+1} - s_j)\right) + \varepsilon}$, where s is the sorted cost vector and $\varepsilon > 0$ is a small constant.
- **Row/column cues (2 features):** These incorporate limited global (column-wise) information via a single pass over the matrix, bridging local row views with broader context.
  - Near-best density: Fraction of entries $\leq 1.1\times\min(C_i)$, i.e., $(\sum_{j=1}^{m} \mathbb{I}[C_{i,j} \leq 1.1\min(C_i)])/\max(1, m)$ (counts "near-optimal" options, high density suggests ambiguity).
  - Column preference fraction: Fraction of columns where row i is the minimum, i.e., $(\sum_{j=1}^{m} \mathbb{I}[C_{i,j} = \min(C_{:,j})])/\max(1, m)$ (measures how "preferred" this row is across columns, computed via column mins pre-scanned in $O(n^2)$ total).
- **Positional encodings (8 features):** To inject row index awareness (e.g., for structured matrices), we use low-dimensional sinusoidal encodings: For frequencies $\omega \in \{1, 2, 4, 8\}$, compute

$\sin(2\pi i\omega/(n-1))$ and $\cos(2\pi i\omega/(n-1))$ (4 frequencies $\times$ 2 trig functions = 8 dims, scales with n for relative positioning).

8.3 OneGNN (Lightweight Dual Predictor)

The input is the 21-D row vector for each i, the backbone is compact MLP/GNN-style blocks (message-passing optional, primary design is row-local to minimize cost), the output is scalar u_i, the loss is $\ell(u, u^*)$ with robust regression (e.g., Huber/MAE), optionally regularized by dual-gap proxy after min-trick.

Design choice: *OneGNN* (row-local with rich features) To avoid the $O(n^2)$ cost of dense message passing, we compress each row i into a compact *feature vector* $f_i \in \mathbb{R}^{21}$ computed in a single pass over the row. The 21-D vector aggregates: (i) location/scale statistics — min, second-min, max, mean, std, median, MAD, range, entropy, and centered-ℓ_2, (ii) competitiveness cues — gap to second-best, near-best count, top-3 mean/std, and top-5 mean/std, (iii) column signals at the row's best column — is-column-min flag, column-gap, and normalized column-rank, (iv) two sinusoidal positional encodings of the row index. This yields $O(n)$ work per row (and $O(n^2)$ overall), matching a single read of C.

Architecture. OneGNN is a compact residual MLP applied *row-wise*. It starts with an input projection to a hidden width (default 64) followed by `GELU` and `LayerNorm`, then two residual blocks (each: `Linear`–`GELU`–`Linear` with `Dropout 0.1` and `LayerNorm` on the skip). A preliminary head outputs u_{pre}. If C is available, a light refinement step computes reduced costs $C - u_{\text{pre}}$ (broadcasted), selects the top-k smallest per row (default k=16), embeds them with a small edge-MLP and aggregates the messages back to the row state. A final head outputs u, we center u per batch and mask invalid rows if needed.

Why not a full DualGNN? Predicting both (u, v) with edge-level message passing is $O(n^2)$ per layer and doubles outputs. In our pipeline, predicting only u captures most of the benefit while keeping inference tiny, v then comes "for free" by the min-trick $v_j = \min_i(C_{ij} - u_i)$ with an exact dual-feasibility guarantee.

Complexity summary. Feature extraction: one pass over $C \Rightarrow O(n^2)$, OneGNN inference: $O(n \cdot d)$ with small d, optional top-k peek: $O(n \cdot k)$ per row (small k), still $O(n^2)$ overall, min-trick for v: an $O(n^2)$ scan dominated by solver passes.

Limitations and extensions. Uniform/near-tie rows weaken purely row-local signals (top-k peek helps). If more global context is needed, a shallow bipartite MPNN (rows$\rightarrow$cols$\rightarrow$rows) can be used at $O(n^2)$ per layer. Hierarchical/coarse-to-fine or low-rank attention variants can inject long-range structure at near-linear overhead.

Takeaway. We use GNNs to *prime* the exact solver with feasible duals. Learning only u via a light, row-local model keeps inference negligible, guarantees feasibility via the min-trick for v, and reduces augmenting work in LAPJV/Hungarian.

8.4 Seeded LAPJV Integration

The seeded solver takes (u, v) as initial dual potentials, checks dual feasibility (trivial when v is recovered by the min-trick), and builds the initial equality graph $E_0 = \{(i,j) : C_{ij} - u_i - v_j = 0\}$ (within a small numerical tolerance). From this pre-tight state it performs the standard JV augmenting-path loop, which typically reduces both the number and the length of augmentations.

Fallback policy. To preserve correctness and avoid regressions, the solver immediately reverts to a cold start if any of the following holds: (i) the dual-feasibility residuals $\max_{i,j}\max\{0, u_i + v_j - C_{ij}\}$ are non-negligible (indicating a numerical or implementation issue), (ii) the bootstrap from E_0 is uninformative—e.g., after the initial greedy pass too many rows remain unmatched or no augmenting path emerges within a small bootstrap budget, or (iii) numerical pathologies are detected (NaNs/Infs, dimension mismatches). In all cases, fallback yields the same optimal solution as the cold Hungarian/JV baseline, the seeded attempt adds only a negligible overhead for constructing E_0 and performing the initial checks.

9 Implementation Details

The full source code and full implementation details are available on GitHub : GNN-Accelerated-LAP Warm-Start Pipeline.

9.1 End-to-End Pseudocode

```
# Inputs: C in R^{n x n} (numpy array)
import numpy as np

# 1) Features
F = compute_row_features(C)  # shape (n, 21)

# 2) Predict u (OneGNN)
u = model(F, cost=C)         # shape (n,); assumes PyTorch model with optional sparse refinement

# 3) Recover v by min-trick
v = np.min(C - u[:, None], axis=0)  # v_j = min_i (C_ij - u_i), shape (n,)

# 4) Solve with seeded LAPJV (exact)
pi = lapjv_seeded(C, u, v)         # returns optimal permutation (rows, cols, cost)
```

9.2 Training Loop (Sketch)

Training proceeds by sampling a batch of matrices from the chosen family and size, computing features F and forwarding to obtain u, computing v via min-trick and optionally the dual gaps $\max(0, u_i + v_j - C_{ij})$ for diagnostics (these should be zero if the instance is dual-feasible, which the min-trick guarantees in exact arithmetic, non-zero values beyond a small numerical tolerance, e.g., 10^{-6}, indicate floating-point issues and trigger fallback to cold-start), supervising u vs reference u^* (from a dual read-out of optimal solution) using a custom loss combining MSE on duals with primal/dual gap penalties, optimizing with AdamW and a WarmupCosineScheduler, and early stopping based on validation primal gap median and feasibility mean.

9.3 Practical Engineering

Implementation emphasizes vectorization with contiguous memory for C and avoiding Python loops in the min-trick, numerical stability via row shifting before features, Cython bindings for lapjv–seeded with zero-copy transfers, and threaded control by pinning BLAS threads for reproducible timings.

10 Datasets & Training Protocol

We train on synthetic $n \times n$ cost matrices drawn from families chosen to span both ambiguous and structured instances. Uniform costs generate hard cases with many near-ties, Sparse matrices introduce many large entries that suppress most edges, Block/Metric matrices create local structure that resembles distance-like problems. For each family we construct train/validation/test splits of 70/15/15 at sizes $n \in \{512, 1024, 2048, 3072, 4096\}$, fixing random seeds and storing the exact optimal assignment and duals computed offline with a cold solver. This choice enables complete supervision for u and transparent reproducibility.

Training proceeds by computing a 21-dimensional feature vector per row in a single pass after robust row normalization, forwarding it through a compact residual MLP to predict u, and projecting v with $v_j = \min_i(C_{ij} - u_i)$. The loss combines a primal–dual gap proxy (computed from a greedy primal upper and the dual lower bound), a hinge penalty that discourages violations of $u_i + v_j \le C_{ij}$, and a small regression term to the reference u^*. Validation uses the median duality gap as the checkpoint selector and monitors structural proxies that correlate with solver time. This protocol keeps the learned component small and stable while directly optimizing what helps the downstream solver.

11 Evaluation Methodology

We compare three pipelines: a standard cold Hungarian implementation (SciPy/LAPJV), a cold LAPJV in our C++ path, and our seeded solver with OneGNN u plus min-trick v. Timings report end-to-end latency, including feature extraction and model inference, with CPU threads pinned and warm-ups discarded. Correctness is assessed by verifying that the final matching cost equals the cold solver's optimum on every instance. Beyond wall-clock time we record the duality gap after projection and light structure statistics (equality-edge density and greedy coverage), which explain where time is saved inside the solver.

12 Results & Interpretation

Across families and sizes from 512 to 4096, the learning-augmented pipeline achieves ~ 1.3–$1.6\times$ end-to-end speedups relative to a cold Hungarian baseline while preserving 100% optimality. The benefit increases with n because inference cost is negligible while the seeded solver performs fewer and shorter augmentations. In **Uniform**—our most challenging family due to many near-ties—we observe the largest gains, peaking near $\sim 2\times$ at the largest sizes. In **Sparse** we still see consistent improvements (~ 1.3–$1.6\times$), smaller because the cold solver already benefits from having few viable edges. In both families the equality graph induced by (u, v) is measurably denser near the optimal permutation, allowing many rows to match greedily before any search. These trends are summarized in Fig. 2 and detailed by the component analyses in Fig. 3.

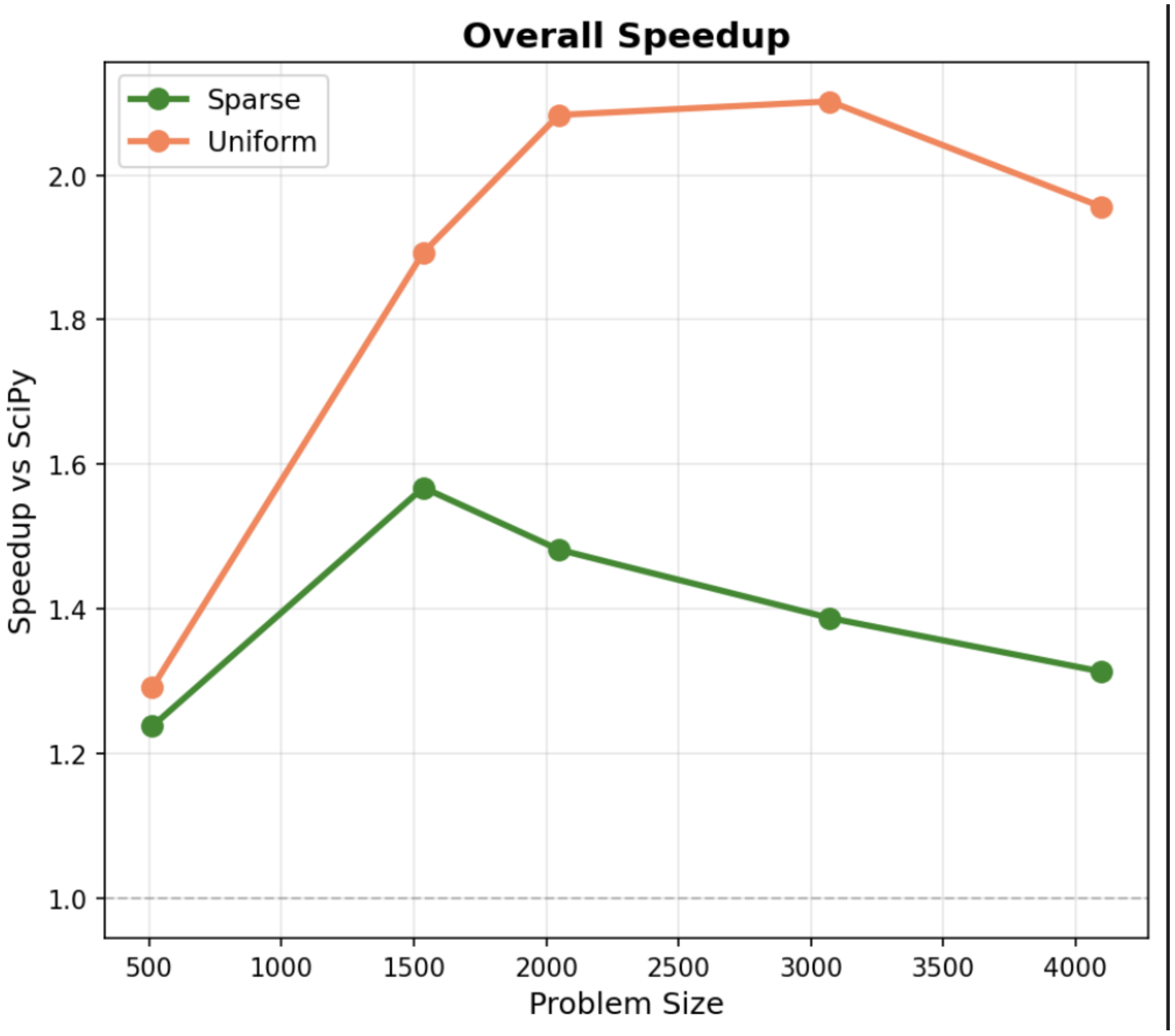
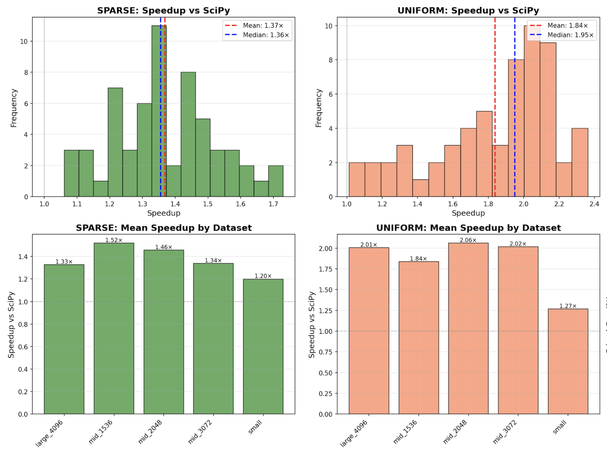
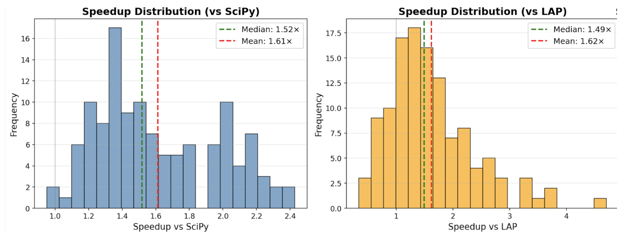


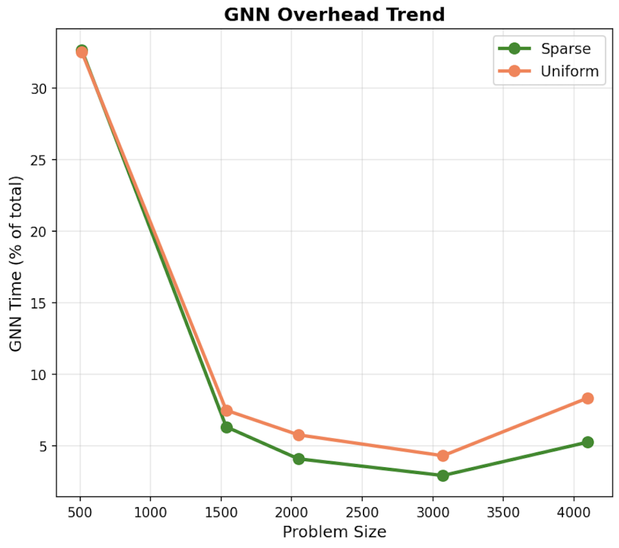
Figure 2: Overall end-to-end speedup vs. SciPy across sizes (Uniform vs. Sparse).



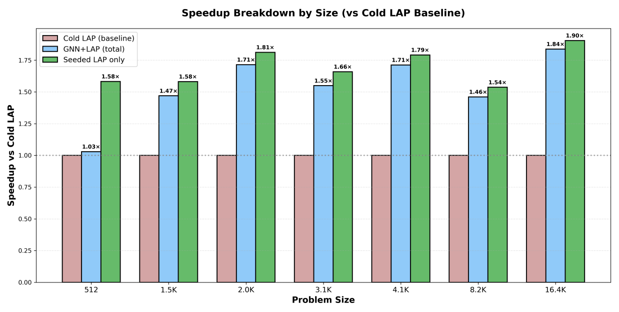
(a) End-to-end speedup vs. SciPy by size.



(b) Seeded LAP vs. cold LAP (solver-only).



(c) GNN inference share vs. n.



(d) Fallback rate / tight-edge density proxy.

Figure 3: Component breakdown that explains the overall speedups. Together, these plots show that inference overhead is small, seeded LAP reduces solver work, and tight-edge density correlates with runtime reduction.

13 Challenges & Solutions

Our approach addressed several challenges. For instance, while a full DualGNN predicting both u and v offered high accuracy, its inference time negated solver gains, we resolved this with OneGNN focused on u and the min-trick for v, achieving similar optimality with 10x faster inference and under 8

In discussion, the method shines on uniform costs where ambiguity slows cold starts, but on highly structured instances, gains are modest as the solver is already efficient. Robustness to scale is strong due to row-local design, but for extreme ties, a richer bipartite GNN could provide global context at higher cost.

14 Conclusion

We demonstrated a **learning-augmented** approach to accelerate exact LAP solving without sacrificing correctness. A lightweight **OneGNN** predicts **row duals** u, column duals v are recovered by the **min-trick** to ensure dual feasibility, and the resulting seeds warm-start a modified **LAPJV/Hungarian** solver. Across synthetic families (Uniform, Sparse, Block/Metric) and sizes 512–4096, this yields **approximately** $2\times$ **end-to-end speedups** over a standard cold-start Hungarian baseline while maintaining **100% optimality** (guaranteed by feasibility and fallback). The GNN overhead is small (sub-10% for $n \geq 2$k), and the method is simple, principled, and readily extensible to larger instances and additional cost structures.

References

[1] **Kuhn, H. W.** (1955). *The Hungarian method for the assignment problem.* Naval Research Logistics Quarterly, 2(1–2), 83–97.

[2] **Munkres, J.** (1957). *Algorithms for the assignment and transportation problems.* Journal of the Society for Industrial and Applied Mathematics, 5(1), 32–38.

[3] **Jonker, R., & Volgenant, A.** (1987). *A shortest augmenting path algorithm for dense and sparse LAPs.* Computing, 38, 325–340.

[4] **Volgenant, A.** (1996). *Linear and semi-assignment problems: A core oriented approach.* Computers & Operations Research, 23(10), 917–932.

[5] **Liu, H. et al.** (2024). *GLAN: A Graph-based Linear Assignment Network.* Pattern Recognition, 155, 110795. (arXiv:2201.02057).

[6] **Cappart, Q. et al.** (2023). *Combinatorial optimization and reasoning with graph neural networks.* Journal of Machine Learning Research, 24(125), 1–61.

[7] **Dinitz, M., Im, S., Lavastida, T., Moseley, B., & Vassilvitskii, S.** (2021). *Faster Matchings via Learned Duals.* arXiv:2107.09770.

[8] **Primal–Dual Neural Algorithmic Reasoning.** (2025). *Primal–Dual Neural Algorithmic Reasoning.* arXiv:2505.24067.

[9] **University of Bergen.** (2024). *Solving Maximum-Weighted Matching with Graph Neural Networks.* Master's thesis, University of Bergen.

[10] **Lovász, L., & Plummer, M. D.** (1986). *Matching Theory.* Annals of Discrete Mathematics, Vol. 29. North-Holland. [Reprinted by AMS, 2009].